ASP-Based Time-Bounded Planning for Logistics Robots

Björn Schäpers, Tim Niemueller, Gerhard Lakemeyer
Knowledge-Based Systems Group,
RWTH Aachen University, Germany
{schaepers, niemueller, lakemeyer}@kbsg.rwth-aachen.de

Martin Gebser, Torsten Schaub
Knowledge Representation and Reasoning Group,
University of Potsdam, Germany
{gebser, torsten}@cs.uni-potsdam.de

Abstract

Manufacturing industries are undergoing a major paradigm shift towards more autonomy. Automated planning and scheduling then becomes a necessity. The Planning and Execution Competition for Logistics Robots in Simulation held at ICAPS is based on this scenario and provides an interesting testbed. However, the posed problem is challenging as also demonstrated by the somewhat weak results in 2017. The domain requires temporal reasoning and dealing with uncertainty. We propose a novel planning system based on Answer Set Programming and the Clingo solver to tackle these problems and incentivize robot cooperation. Our results show a significant performance improvement, both, in terms of lowering computational requirements and better game metrics.

1 Introduction

Industrial applications move towards more autonomy in smart factories, context-aware facilities that assist in the execution of manufacturing tasks. This scenario is modeled by the Planning and Execution Competition for Logistics Robots in Simulation (PExC) (Niemueller et al. 2016) for a small fleet of three robots maintaining and optimizing the material flow in a virtual factory with six machines. While obviously planning is useful to accomplish this task, it is still the exception rather than the norm. The performance in the first competition in 2017 was subpar. Investigating the domain, several challenges arise. Even for the small number of objects, the combinatorial growth quickly becomes overwhelming. This is only made worse when the temporal aspects inherent to the domain are fully modeled. For instance, orders must be delivered in specific time windows and are announced only at run-time with a certain lead time. Even more severe, there is uncertainty, e.g., in terms of time required to move in-between locations due to two teams competing at the same time, or due to processing stations being down for maintenance for limited (a priori unknown) time.

Our main contribution is a pattern for modeling time-bounded temporal multi-robot planning with Answer Set Programming (ASP) (Brewka, Eiter, and Truszczyński 2011), which enables us to use the full power of ASP to steer the search. As our planning approach is time-bounded, it generates actions for a specified time window (e.g., for the next three minutes). Planning is considered successful if a partial plan can be found, even if no full goal state is reachable within the time window. To this end, domains have to provide an evaluation metric that accumulates within the time window in order to ensure eventual progress towards the goal. Our planner then generates temporal plans with concurrency for efficient multi-robot (co)operation, making use of Clingo's multi-shot solving (Gebser et al. 2014) for virtually continuous and adaptable solving. We thoroughly evaluate our approach by comparing it to the POPF planner (Coles et al. 2010) and by simulating PExC games against publicly available agents, among them the 2017 PExC winner as well as the winner of the 2016 RoboCup competition.

In the following Section 2, we give a brief overview of related work and this work's background, before explaining our approach in detail in Section 3. We present our evaluation results in Section 4 before concluding in Section 5.

2 Related Work and Background

We briefly introduce our evaluation domain, related planners such as POPF and Plasp, and Answer Set Programming.

2.1 Planning and Execution Competition for Logistics Robots in Simulation

The Planning and Execution Competition for Logistics Robots in Simulation (PExC) (Niemueller et al. 2016) is based on the RoboCup Logistics League (RCLL) (Niemueller, Lakemeyer, and Ferrein 2015). It considers

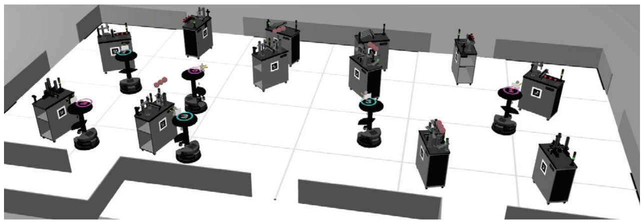

Figure 1: The PExC/RCLL simulation environment.

two teams of three robots each that need to maintain production according to a dynamic order schedule. A product consists of a base (out of three possible colors), zero or up to three color-coded and ordered rings, and a cap (out of two colors). Products are assigned a complexity C_0 to C_3 depending on the number of rings. Orders specify the requested product and a time window when the product can be delivered. There are six stationary machines per team on the field (at a priori unknown positions), which provide specific functionality, such as supplying bases of specified colors, mounting a ring or a cap, or accepting deliveries of completed products. Orders become known only at run-time with a certain lead time before the beginning of the delivery time window. The teams are oversubscribed, i.e., the order schedule asks for more products than can be produced and delivered in the given time of 15 minutes. Therefore, teams must dynamically decide which orders to pursue, what production steps are necessary, and how to schedule this to the available robots. Uncertainty arises through the presence of a second team, which can delay robots while driving (and avoiding collisions), and because machines may be in maintenance and thus unavailable for limited time.

2.2 POPF

POPF (Coles et al. 2010) is a temporal planner based on PDDL 2.1 (Fox and Long 2003). It aims at balancing total order planning, as done by most forward-chaining planners, and partial order planning as an "intuitively attractive strategy". While forward-chaining commits to an action's time frame as soon as the action is added to a plan, partial order planning pursues the idea of least commitment and delays setting actual times for planned actions as long as possible.

POPF's approach is committing to the order of the added actions, in regard of used or modified facts and numerics. That is, adding an action A with p in its precondition, the action A has to start after the last known adder of p. Likewise can another action B that would remove p at its beginning only start after action A is finished. To adhere to these constraints, POPF checks when a fact was last added, removed, and when it has to hold. The commitment to the timing of unrelated actions is delayed. Moreover, POPF is a complete planner because, if a poor choice of ordering occurs, it backtracks and continues to calculate a new plan.

2.3 Answer Set Programming

Answer Set Programming (ASP) (Brewka, Eiter, and Truszczyński 2011) is a declarative programming language with stable model (answer set) semantics of logic programming (Lifschitz 2008) and a similar syntax to specify rules. The idea is to express a problem in logical format so that models of its representation provide solutions to the original problem. Hence, we formulate our planning task by an ASP encoding and read off the resulting plan from an answer set.

An ASP encoding is made up of rules of the basic form $head \leftarrow body$, e.g., $r = a_0 \leftarrow a_1, ..., a_n, {\sim}a_{n+1}, ..., {\sim}a_m$, with atoms a_i for $i \in \{0, ..., m\}$. A rule with an empty body is a fact, and a program is a collection of rules. Two special forms of rules are of particular relevance. First, for a *choice rule* of the form $l\{a_1, ..., a_n\}u \leftarrow a_{n+1}, ..., a_m, {\sim}a_{m+1}, ..., {\sim}a_o$, a solver can add any subset of $\{a_1, ..., a_n\}$ to the answer set, provided that at least l and at most u elements are added. In our planner, such choice rules represent the possible actions for a plan. Second, *integrity constraints*, rules with an empty head, prune answer sets in which the body of such a rule holds. This permits, for example, preventing two robots from performing the same task.

Searching for answer sets is done in two steps (Kaufmann et al. 2016). First, the *grounder* instantiates an ASP encoding through variable substitution to create ground instances of the rules. Second, the output of the grounder is processed by the *solver* to find answer sets. This can involve optimization according to some metric. We use Clingo (Gebser et al. 2016), which integrates both aspects into a common framework. In particular, we harness *multi-shot solving* (Gebser et al. 2014) for iterative grounding and solving, specifically appealing to incomplete knowledge, e.g., regarding orders yet to be announced. Clingo supports specifying multiple program parts that can be grounded separately. Furthermore, and more importantly, it supports *externals*, i.e., undefined atoms that may change in-between solving iterations.

2.4 Plasp

Plasp is a system for planning by compilation of a PDDL specification to ASP (Gebser et al. 2011). While this translation is simple, it supports modeling different planning techniques in ASP by meta-programming. The ASP grounder is used to obtain propositional representations and the solver to retrieve actual plans. Plasp so far only supports the STRIPS subset of PDDL and can thus not handle temporal domains.

We have run experiments with Plasp for PExC, where the domain was simplified to a single robot, we ignored all temporal aspects, and the task consisted of a single C_0 order without any rings.[1] Even for this simple setup, Plasp was unable to find a solution within an hour. Plasp also is not able to operate on-line, i.e., it builds on the assumption of complete information. Since the capabilities and performance of Plasp turned out as insufficient, we will not consider PDDL-based ASP planners in the remainder of this paper.

3 ASP-based Time-Bounded Planning

As illustrated in Figure 2, we employ the paradigm of *centralized, global planning*, where a single computer plans for the full fleet of robots (Niemueller, Lakemeyer, and Ferrein 2015). On this central computer, a wrapper collects all its data through a shared database (robot memory) to provide the necessary information to the planner. It also controls Clingo by instructing when to ground or solve. A plan is then extracted from an answer set (if any) and passed on to the robots through the shared database. On the robots, a controller selects actions assigned to the specific robot adhering to its timing requirements and executes them. The robots in turn provide all necessary information back to the planner host. In particular, they report when executing an action takes longer than anticipated, in which case the planner may decide to replan. Our system is based on a publicly available

[1]We also removed ring station related specification parts, as the system would run out of memory at 16 GiB of RAM otherwise.

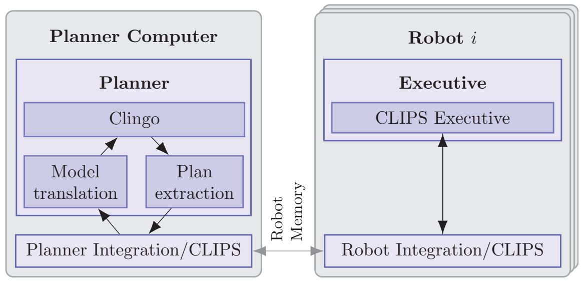

Figure 2: Planning system architecture. Grey boxes represent computing devices, blue boxes components and subcomponents, and arrows indicate the flow of information.

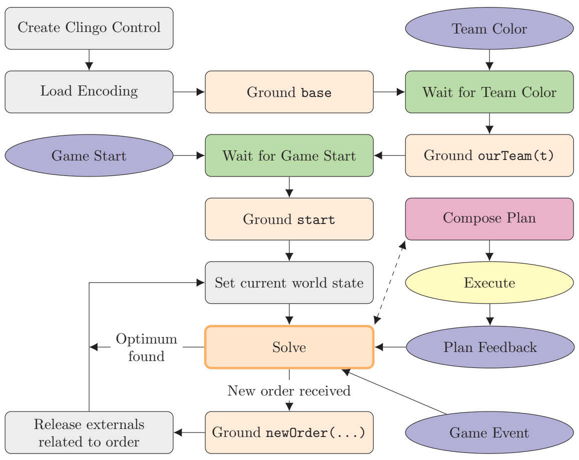

Figure 3: Program flow diagram. The main planning and execution cycle is included on the right. We show only the event of a new order as an example for interrupting planning.

implementation using the CLIPS rules engine (Niemueller, Lakemeyer, and Ferrein 2013),[2] and integrating our own planning system required only modest modifications.

3.1 Solution Overview

The domain is encoded directly in ASP, where tasks can be added to an answer set through choice rules. The overall encoding is split into several parts, which can be grounded separately. The general planning procedure at a glance is depicted in Figure 3. Through multi-shot solving, we resolve based on updated information on specific events, such as a new order coming, without requiring costly grounding of the whole program again. Special external atoms permit updating information for the next solving call, and a resulting answer set contains specific atoms yielding a plan. In the following, we describe these constituents in more detail.

3.2 Encoding Idea

Unlike Plasp, for example, which reads a PDDL description, encoding the domain directly in ASP avoids compromises in

[2]https://www.fawkesrobotics.org/p/rcll2016-release/

1 $robot(1) \leftarrow$
2 $robot(2) \leftarrow$
3 $at(1, l_1, 0) \leftarrow$
4 $at(2, l_3, 0) \leftarrow$
5 $\leftarrow {\sim}at(1, l_2, q)$
6 $\leftarrow {\sim}at(2, l_4, q)$
7 $task(drive(L)) \leftarrow location(L)$
8 $at(R, L, T) \leftarrow end(R, drive(L), T)$
9 $at(R, L, T) \leftarrow at(R, L, T-1), {\sim}begin(R, \_, T-1)$
10 $\{begin(R, A, T) : task(A)\}\, 1 \leftarrow robot(R),\, T = 0..(q-1)$
11 $do(R, A, T, D) \leftarrow begin(R, A, T), duration(R, A, T, D)$
12 $do(R, A, T, D) \leftarrow do(R, A, T-1, D+1), D > 0$
13 $end(R, A, T) \leftarrow do(R, A, T-1, 1)$
14 $\leftarrow do(R, \_, T-1, D+1), D > 0, begin(R, \_, T)$
15 $\leftarrow do(R_1, A, T, \_), do(R_2, A, T, \_), R_1 \neq R_2$

Logic Program 1: Example encoding for path finding.

terms of efficiency and solving capabilities, as it increases modeling flexibility, especially regarding the representation of constraints. Nevertheless, our approach yields a general pattern for modeling temporal planning problems that is independent of any other description language.[3]

Our systems plans over tasks, rather than primitive actions. *Tasks* are similar to *macro actions* that treat several primitive actions as a compound to improve search efficiency. In the following, we will use action and task interchangeably. For an action A of a robot R at time T with a remaining duration of D, the begin and end times are represented by $begin(R, A, T)$ and $end(R, A, T)$ atoms, while atoms of the form $do(R, A, T, D)$ keep track of an active task. An example is shown in Logic Program 1, where the encoding is split into a domain-specific (ll. 1–9) and a general part (ll. 10–15). Lines 1–4 state facts about the initial situation, i.e., there are two robots, and robot 1 is at location l_1 and robot 2 at location l_3 at time 0. The goal is expressed by constraint rules (ll. 5–6), i.e., robot 1 has to be at position l_2 and robot 2 at position l_4, where q represents the planning horizon giving the latest time when the locations should be reached. There is a single kind of task to drive to a certain location (l. 7). Note that actions have no direct argument denoting which robot is supposed to execute the task, as this is handled by the generic encoding part. In our example, we assume that driving is always possible, while an atom *possible* were used to check actions' preconditions otherwise. The domain-specific effect of performing a drive action is changing the robot's location (l. 8), or we keep the robot's last position in case the robot is idle (l. 9).

In the generic part, line 10 provides a choice rule that allows for picking tasks for specific robots by adding a *begin* atom to an answer set. Lines 11–13 describe the bookkeeping for performed actions, where a *do* atom is derived at the time a task starts and then counts the remaining time. The constraint in line 14 makes sure that no (other) action is started by a robot as long as the execution of the previous task is still ongoing. Finally, line 15 expresses that any task can be performed by at most one robot at a given time.

[3]For instance, Plasp could be extended to take PDDL input and translate it into the very pattern we formulate.

```
getBaseTask(getBase(L,B))             :- baseLocation(L), baseColor(B).
task(T)                               :- getBaseTask(T).
%points(T, 10)                        :- getBaseTask(T).
taskBase(getBase(L,B), B)             :- task(getBase(L,B)).
taskDuration(T, @getTaskDuration())   :- getBaseTask(T).
taskLocation(getBase(L,B), L)         :- task(getBase(L,B)).

toBeDone(T,GT)    :- getBaseTask(T), productionStarted, horizon(H), GT=0..H-1.
inUse(L,R,GT)     :- doing(R,T,_,GT), getBaseTask(T), baseLocation(L).
inUse(L,R,GT)     :- end(R,T,GT), getBaseTask(T), baseLocation(L).
possible(R,T,GT)  :- getBaseTask(T), robot(R), not holding(R,_,GT), horizon(H), GT=0..H-1.

:- doing(R1,T1,_,GT), doing(R2,T2,_,GT), getBaseTask(T1), getBaseTask(T2), R1 != R2.

generateProduct(R,B,GT) :- end(R,T,GT), getBaseTask(T), taskBase(T,B).
pickUp(R,P,GT)          :- end(R,T,GT), getBaseTask(T), generatedProduct(R,P,GT).
```

Listing 1: Encoding of the task to get a base from the base station.
(L: location, B: base color, T: task, H: horizon, GT: game time, R: robot, P: product/workpiece)

Let the durations from l_1 to l_2 be 3 s, and from l_3 to l_4 5 s. A possible answer set may then include the following atoms for $q \geq 5$: $begin(1, drive(l_2), 0)$, $end(1, drive(l_2), 3)$, $begin(2, drive(l_4), 0)$, and $end(2, drive(l_4), 5)$. Such atoms provide a plan and implicitly determine other atoms like at.

3.3 Time Representation and Time Bound

In our previous example, we assume time to be represented by a natural number. However, in the PExC domain, where games are 900 s long, this leads to a large blow-up of the state space. Therefore, we *discretize time into intervals*.

We have already specified a *planning horizon* by means of the query time q. In the encoding, we specify this horizon explicitly. Only within this time window will the planner create and optimize a plan. That bounds the search further to enable the anticipated frequent replanning cycles as new information becomes available. For the search to reach the intended goal eventually, the chosen *optimization metric must ensure progress* (e.g., reward) within the time horizon.

We have run several hundred games to *calibrate these parameters*. As is to be expected, the parameters are not independent, e.g., similar solving time for a larger horizon can be achieved by reducing the time resolution. However, this leads to more idle time (actions are only planned to start at interval bounds) and thus decreases task performance. For the PExC domain, we have chosen an interval length of 10 s and a planning horizon of 180 s.

3.4 ASP Domain Encoding

Our encoding of the PExC domain is split into several components, dealing with the general planning mechanics, rules of the game, and robot tasks. The overall encoding comprises less than 500 lines of code.

Our encoding is influenced by the capabilities of the Clingo solver, in particular, multi-shot solving introduced with Clingo 4 (Gebser et al. 2014). It supports separating an encoding into multiple parametrized parts with the **#program** directive, which can be grounded independently of each other. The components mentioned before separate concerns in terms of the domain mechanics and their modeling, and thus help the domain designer to structure the encoding. Using multiple programs, where a program is often mentioned in more than one component, separates parts of the encoding with the same grounding requirements. Our encoding contains 6 programs. The implicit `base` program contains basic facts and domain-specific (but non-team-specific) externals (see below). It is grounded on startup of the planner. A program `ourTeam(t)` (for `t` being the team's jersey color) is the largest program. It contains all information that is specific to a particular team and grounded once the jersey color is known. Three further programs `newOrder`, `setDriveDuration`, and `setRingInfo`, are used by the planner integration plugin to pass information to the solver as it becomes available, i.e., once an order request has been received, or once the production phase starts and information such as the position of the machines and the configuration of the ring stations becomes available. Finally, the program `start` is used to indicate the start of the game.

Updating information in-between grounding and solving iterations is accomplished by means of *externals*. These are special *input atoms* marked with the **#external** directive. These atoms are exempt from simplification and pruning by the solver. Externals are necessary to capture incomplete and volatile information in the domain. For example, the external `base(O,B): order(O), baseColor(B)` describes the base color `B` of an order `O`. Specifying this as an external prevents the solver from removing rules related to orders during initial solving when no orders have been announced, yet. In this way, the expected number of orders and the possible base colors constitute static information. Later, the `newOrder` program is used to ground information about a specific order and the externals for that order can be released. An interesting property in this respect is that solving speed improves over time as the number of externals is reduced as more information becomes available. Other externals such as `availableRobot(R): robot(R)` that describes whether a robot `R` is currently alive and thus available to perform tasks, are never released but merely used to provide volatile information about the world state.

One particular issue when modeling the PExC domain is the requirement of exclusive access to machines, i.e., only one robot may operate a machine at a time.[4] We therefore discretize the space of locations to just the ones of interest, i.e., one on either side of a machine and starting positions. Then, each task is explicitly assigned a specific location. While a robot is traveling, it is assumed to be at no specific location. Listing 1 shows an example of an encoding modeling a task to retrieve a new colored base from the base station. In lines 1–2, we declare the `getBase` task, and line 3 shows how points would be associated with a task (getting bases does not actually award points, hence this is commented out). In the following, we fix the color parameter (l. 4), the task duration (l. 5, where the `@` indicates an external function call) and the location of the task (l. 6). Then, we deduce the task as to be executed (l. 8) and mark a location as being occupied by a robot while executing the task (ll. 9–10), before specifying the task's preconditions that the robot may not be holding anything (l. 11). The constraint in line 13 expresses that at most one robot can perform a `getBase` task at a time.[5] Finally we add information about the generated product and the robot that picked up the product (ll. 15–16). Similar encodings are defined for the remaining tasks, such as fetching or delivering a workpiece, or moving from one place to another.

3.5 Goal and Optimization

The robots are generally *oversubscribed*, that is, there are more orders than can actually be fulfilled. Hence, the system must choose which orders to pursue. Unlike classical (and also temporal) planning systems that require a goal to be specified before starting to plan, we *only need to specify an optimization metric* that maximizes the score (cf. Listing 2). Through the formulation of the scores of the tasks, the planner will eventually produce and deliver goods. However, our system schedules task for various orders very densely (cf. Section 4). While PDDL3 (Gerevini and Long 2005) does support formulating soft and hard goals (and therefore could support automatic goal selection), it only considers solutions which contain the full sequence from the initial situation to at least some goals. In our system, however, feasible (and sometimes even optimal) models may contain partial plans. This is because we only plan with a limited time horizon and do not search towards a goal, but maximize our metric within the given time window.

As a basis, we have used the scoring scheme of the competition. We then made it more fine grained for the ring scores. For example, according to the rules mounting the third ring of a C_3 product scores 80 points. In our model, however, we award 10, 20, and 50 for the first, second, and third ring respectively (for any complexity). This is to ensure the required metric progress (cf. Section 3.3). Otherwise, the planner may not make progress on more complex products within a time window as this would require completing all three ring mounting operations.

[4]Two robots can still cooperate. For example, a robot may retrieve a workpiece prepared by another robot.

[5]The base station can serve bases on both sides, but only one at a time. Other tasks do not have a similar restriction.

```
#program ourTeam(t).
#maximize{P, T : end(_,T,GT), points(T,P)}.
```

Listing 2: Optimization statement for the solver.

3.6 Multi-Shot Solving and Execution

We use Clingo 5 (Gebser et al. 2016) as solver. We rely in particular on its multi-shot solving capabilities by means of programs and externals (cf. Section 3.4). The base program is grounded upon startup, while the team program is grounded during the setup phase of the game, a 90 s period where teams can prepare before the actual game starts. At that time, the team knows about its jersey color. Initial grounding is a costly process and can take up to a minute.

The core of the task selection of the planning procedure is shown in Listing 3. As part of the team specific program (l. 1), the solver may choose one action per robot (ll. 2–3). Several constraints require that a robot is at a location, i.e., not currently driving (l. 4), the task is yet to be done (l. 5) and possible (l. 6), and we start it as early as possible (ll. 7–9).

Once the planner is started, it operates in an *any-time* fashion. That is, during the game, we are *running the solver virtually all the time*, and stop it on specific events, such as a new order being announced, or a machine coming back from maintenance. Once the new information is incorporated the planning process is restarted with the current status. When a new answer set is found, we wait for half a second for stabilization (especially when starting a new solve call the first two or three models may appear in quick succession) and then accept the answer set. The solver continues thus making it possible to find improved answer sets.

The planner *extracts the plan from the accepted answer set*, that is, it determines tasks, the assigned robot, and when the task is supposed to be executed. The new plan is validated with the currently running plan. If the old and new plans are compatible, the new plan is deployed. Conflicts may arise since the old plan keeps being executed when a new solving iteration is run. A robot might have completed a task and started another one, which the new plan now had assigned differently (or not at all). Then, replanning is initiated and the old plan remains in effect. Principally, solving could stop if the solver found an optimal (time-bounded) solution. However, in our experiments this does not happen.

```
1 #program ourTeam(t).
2 { begin(R,T,GT) : task(T) } 1 :-
3       robot(R), horizon(H), GT = 0..H-1.
4 :- begin(R,_,GT), not robotLocation(R,_,GT).
5 :- begin(_,T,GT), not toBeDone(T,GT).
6 :- begin(R,T,GT), not possible(R,T,GT).
7 :- begin(R,T,GT), possible(R,T,GT-1),
8                   robotLocation(R,_,GT-1),
9                   toBeDone(T,GT-1).
```

Listing 3: Task selection choice and constraint rules.

Rather, we restart in the mentioned situations.

The *plan is deployed through a shared database* to which all robots have access. Whenever a robot has completed a step, it picks up the next assigned task and executes it. It keeps track of the actual and the expected time taken. If discrepancies exceed a given threshold, it notifies the planner, which will replan (not necessarily, but possibly, coming to a new solution).

3.7 Working with intermediate results

The usage of intermediate results and partial plans is critical for the scenario. The highest scoring products, C_3, can not be produced within the 3 minute time window, but as we award the internal score for steps towards this goal the tasks are scheduled anyway. And when the solving is restarted some of the work will be done already, up to the point where the C_3 delivery is schedulable within the time frame. On the other hand increasing the horizon so that the full production can be scheduled within one frame, the solving time to find an answer set containing the delivery is so long, the solving will most likely be stopped before finding the answer set.

New plans are immediately deployed, given they are compatible. The optimization value of new answer sets is always higher than the one before so the resulting plan is better according to our metric.

4 Evaluation

To assess the feasibility and performance of our approach we have conducted an in-depth evaluation. In Section 4.1, we compare our planning system to POPF on generated problems. We then describe our evaluations to guide some of our design and parametrization decisions in Section 4.2. We have run a large number of simulated test games which we describe in Section 4.3. All evaluation was performed on a cluster of 7 machines consisting of two with Intel Core i7-6700 and five with i7-3770 quad core CPUs (all running at $3.4\,\mathrm{GHz}$ with $16\,\mathrm{GiB}$ of RAM). We used safe compiler settings compatible with both types of machines.

4.1 Comparison with POPF

To get an impression of the performance of our planning system, we have run a large number of problems with randomized machine position and order configurations. We have chosen POPF as a representative of PDDL-based temporal planners.[6] For one, it performed well in International Planning Competition (IPC) 2011, for another it comes readily integrated with ROSPlan used in the on-line evaluation.

We have based our POPF tests on the domain which is part of the PExC reference implementation[7] that uses POPF via ROSPlan. Then, we have generated 100 game instances based on the order distribution and random machine positions as specified by the PExC referee box. For each of these game instances, we have created 10 problems for single order production of each complexity for a single or three robots, and problems for the full set of orders (assuming all orders to be known at start). For C_0, we made a strict requirement to produce plans within $60\,\mathrm{s}$, which is the outmost acceptable value for actual games. We also use this configuration during the on-line evaluation with ROSPlan. For the more complex products, we permit a maximum planning time of $30\,\mathrm{min}$, similar to the IPC. For ASP, we have run the planner for the full game duration (F), and with our time-window approach (W). There, we used a time interval of $10\,\mathrm{s}$ and planned every $30\,\mathrm{s}$ for the next $180\,\mathrm{s}$. We accumulate solve time among all solve steps. We have run POPF in its any-time (A) mode, In this mode, POPF continues solving until a timeout is reached and it is aborted. The solving time then is the run-time. Note that POPF minimizes makespan of the plan, while ASP-based time-bounded planning (ASP-TBP) maximizes score. However, a previous analysis (Niemueller et al. 2015) has shown that in this specific domain these amount to the same optimal outcome.

Table 1 summarizes the results. Over all, ASP-TBP is able to solve about 82 % of the problems. The time window approach can solve all. POPF can solve about 16 % of the problems. Using the time-bounded planning approach has clear advantages. It is able to solve all problems.[8] For single orders, the planning times are well within the acceptable range for PExC. Planning a full set of orders from the start takes a very long time. However, if that time is spread over the whole duration of the game (as is done for actual games in the Section 4.3), it still yields acceptable times. Recall that the given time is the sum of all 24 planning iterations,[9] resulting in $17\,\mathrm{s}$ and $22\,\mathrm{s}$ per-cycle planning time averages respectively. The makespan for the ASP approach is considerably longer. This is mostly due to the time discretization. If a task has a duration of $7\,\mathrm{s}$, it is summed as $10\,\mathrm{s}$ in the makespan. Furthermore, the ASP agent has a more tidy modeling. For one specific production step, pre-filling a cap station, the robot ends up with one scrap item. The PDDL model supports simply dropping the item on the floor, while the ASP encoding always recycles.[10]

For all higher order complexities, POPF runs out of memory for all problem instances at $4\,\mathrm{GiB}$. For some, it still creates a plan before aborting, which we count as a successful run. This is due to the fact that we run the 32-bit version coming with ROSPlan. Experiments with a 64-bit version showed worse performance in the solvable instances in terms of solving time and memory consumption, so we did not run this version on the full set.

The maximum memory consumption of Clingo was $5.8\,\mathrm{GiB}$ for 3 robots for a C_3, in full game mode. For a single robot and a C_0 product, Clingo requires about $90\,\mathrm{MiB}$

[6] We have run Clingo with a single thread to increase similarity with POPF, which does not support parallelization.

[7] https://github.com/timn/ros-rcll_ros

[8] To solve a problem means delivering the product for the C_0 to C_3 cases, and delivering at least one product for the full set case. Although in some of the 100 full set problems all products were delivered, this is only possible because robots are assumed to move very fast for the sake of this experiment.

[9] The last planning iteration starts at $12\,\mathrm{min}$ and covers everything until the end.

[10] Note that the makespan and scores are not directly comparable to the on-line evaluation, as for simplicity the given model assumes an unrealistic constant travel speed of $1\,\mathrm{m/s}$ speed. ASP and POPF results are comparable since they use the same time model.

		One robot			Three robots		
		ASP F	ASP W	POPF A	ASP F	ASP W	POPF A
C_0	# solved	**100**	**100**	95	73	**100**	10
	makespan Median (s)	260	240	**120.88**	140	175	**121.67**
	solve time Median (s)	12.25	**1.52**	60.06	24.44	**1.79**	60.05
	first model Median (s)	11.3		**4.89**	**20.65**		27.71
	points Median	36	**42**	30	36	**42**	30
C_1	# solved	**100**	**100**	25	99	**100**	0
	makespan Median (s)	335	260	**172.49**	250	**210**	
	solve time Median (s)	126.75	**2.87**	318.66	374.51	**5.08**	
	first model Median (s)	91.13		**79.26**	**363.1**		
	points Median	55.5	**57**	52	55	**57**	
C_2	# solved	**100**	**100**	21	53	**100**	0
	makespan Median (s)	420	380	**213**	310	**250**	
	solve time Median (s)	285.09	**5.87**	344.16	852.94	**35.41**	
	first model Median (s)	257.9		**103.17**	**748.1**		
	points Median	**89**	**89**	77	83	**89**	
C_3	# solved	96	**100**	10	21	**100**	0
	makespan Median (s)	490	445	**252.35**	350	**290**	
	solve time Median (s)	552.7	**7.12**	368.1	1178.09	**78.97**	
	first model Median (s)	534.04		**145.81**	**1177.66**		
	points Median	152	**157**	144	151	**156**	
Full Set	# solved	0	**100**	0	0	**100**	0
	makespan Median (s)		**830**			**640**	
	solve time Median (s)		**405.75**			**524.99**	
	first model Median (s)						
	points Median		**268**			**391**	

Table 1: Comparison of ASP Planning with POPF. For each robot (macro column) and order configuration (macro rows) there are 100 problem instances, resulting in $2 \times 5 \times 100 = 1000$ runs in total. Bold face entries mark the best entries per configuration (one per line and per number of robots).

in time window mode. The consumption of a typical online configuration with three robots, all orders, and using the time window approach is about 250 MiB.

4.2 Solver and Encoding Parametrization

During the development of the ASP-based planner, we have run several hundred games to guide the design of our encoding in terms of planning and metric performance, as well as to inform our decision of solving parameters.

One aspect of the encoding we have investigated in detail is the way we *model movement* from one location to another (the "goto" task). If the encoding allows the robot to move among locations without further constraints, plans tend to be very inefficient with many consecutive useless gotos. This is because we optimize score, on which a goto has no impact. This is a problem that can also often be observed with PDDL domain models. To improve this, we then added constraints to force a robot to perform a task, when one was possible at its current location. This removed most useless movements. While gotos and some other tasks have no direct impact on the score, neither positive nor negative, their execution costs time which could be used to perform actions which yield score. Given enough time the solver would always find the optimal answer set, not containing any useless gotos. But as time is crucial we have to guide the solver with the additional constraints to find better answer sets faster.

As described earlier in Section 3.3, we have systematically tried several combinations of time interval length and planning horizon. A *horizon* of 60 s performed poorly, while 180 s was much better. No further improvement could be reached by increasing the horizon; to the contrary a horizon of 300 s or more showed deteriorated performance due to long planning times and high memory consumption due to a much larger grounding. Using a *time intervals* of 7 s required too much memory. Using 13 s did not show a significant improvement over 10 s but risks higher robot idle times. We therefore went with a horizon of 180 s at a 10 s time interval.

Clingo can be run with multiple threads in several *threading modes* for parallel search during solving. In split mode, the search space is separated into disjunct pieces and threads search exclusively in their sub-spaces. In compete mode, each thread operates on the whole search space but with different search strategies. We have tried both modes with 2, 4, and 7 threads (hyper-threading supports 8 parallel threads, we left one free for the planner integration). Increasing the number of threads has severe impact on memory requirements. After many test runs we settled for running with two threads in split mode.

While the solver and encoding parameters have been determined empirically, future work could investigate the (partially) automated determination of these values.

4.3 Simulated Tournament

We have based our evaluation in actual games on the publicly available cluster setup[11] of the Planning and Execution

[11]https://github.com/timn/rcll-sim-cluster

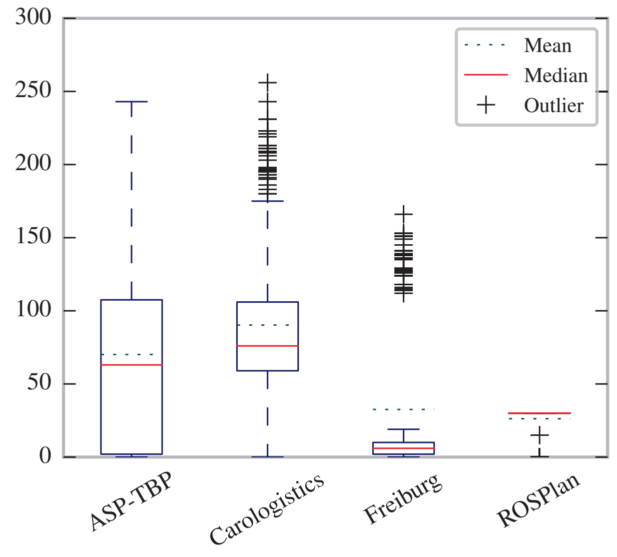

Figure 4: Box plot of scores achieved in 600 simulated. A box shows the 25% and 75% quartiles (lower and upper bound of box), min and max points within expected tolerance (whiskers with caps), median, mean, and outliers. An outlier is a score with a difference of more than 1.5× the inter-quartile distance from the nearest bound.

Competition for Logistics Robots in Simulation[12] (PExC) at ICAPS 2017. We have run a competition with the publicly available CLIPS-Agent[13] (Niemueller, Lakemeyer, and Ferrein 2013), ROSPlan[13] (Cashmore et al. 2015), and the winning team from Freiburg.[14]

For each team combination, we have run 100 games, resulting in a total of 600 games. The CLIPS-Agent won 232 games, our ASP planner won 179 times, and Freiburg and ROSPlan won 93 and 85 times respectively (each system played 300 games). Figure 4 shows a box plot of the resulting score distribution. The CLIPS-agent has consistent scoring (narrow box) and the highest median score and many upward outliers. Freiburg does not perform as robust, but still manages to score high in many games. The baseline ROSPlan implementation only pursues a single C_0 product, which it most of the time is able to finish. Our approach scores reliably (indicated by the high median). However, the lower quartile is really low, resulting in more opportunities for other teams to win. The upper quartile is even higher than the one of the CLIPS-Agent. So further improvements to robustness of planning and execution could provide the necessary edge to supersede the CLIPS-Agent. Note that the CLIPS-Agent does not do any planning, rather it performs situation classification and then selects the next best action whenever a robot is idle. The CLIPS domain modeling consists of more than 4000 lines, encoding very specific situations. While this does perform well, it required years of development for the domain modeling and makes it tedious to encode cooperative behavior and coordination.

[12]https://www.robocup-logistics.org/sim-comp

[13]https://github.com/timn/docker-robotics

[14]https://github.com/GKIFreiburg/rcll-sim-freiburg

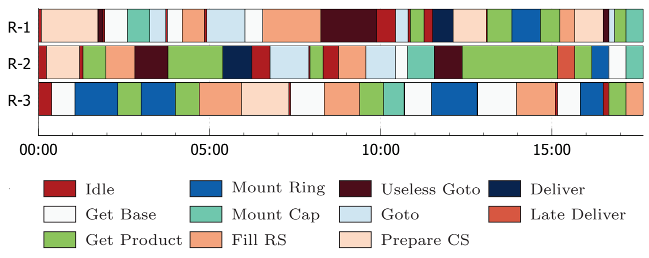

Figure 5: Gantt chart of robot task assignments for the robots R-1 to R-3 with the time denoted in minutes.

Figure 5 shows the task assignment to three robots in a game of the competition. At a first glance, in particular, the dense task assignment is obvious. The plan is generated up to the full horizon ignoring the end of a game (and hence actions reach beyond 15 min. The time of a task also includes driving to the location where the task must be performed. The explicit goto allows robots to move to a location in anticipation of a task that needs to be done there later. The initial idle time is when the robots enter the field. In this instance, R-3 immediately gets a base, while R-1 and R-2 start by preparing the two cap stations. This is a typical game of our ASP-based system scoring 117 points.

5 Conclusion

We have presented a novel approach to ASP-based time-bounded planning. It uses a direct encoding of the planning domain and procedure in ASP, and the Clingo grounder and solver to find plans. Using improvements such as macro actions, time-bounded planning, and multi-shot solving yields the required efficiency to be competitive in the Planning and Execution Competition for Logistics Robots (PExC). We have compared our approach to the POPF planner, where our planner is able to solve many more problems in much shorter planning times. This translates into very successful runs based on the publicly available PExC cluster setup. We could outperform the available planner integrations and the performance is close to the CLIPS-based reference system, but with an encoding size an order of magnitude smaller, fully automated, and anticipating potential for cooperation automatically.

Further improvements could be made in terms of robustness, i.e., increasing the lower scoring quartile. Future work could also be directed towards generalizing the concepts and ideas and performance gains of our system, for example by extending Plasp.

The source code and evaluation data are available at https://www.fawkesrobotics.org/p/asp-tbp.

Acknowledgments

T. Niemueller and B. Schäpers were supported by the German National Science Foundation (DFG) research unit

FOR 1513 on Hybrid Reasoning for Intelligent Systems (http://www.hybrid-reasoning.org).

We thank the anonymous reviewers for their insightful comments and questions which helped clarify several aspects of this paper.

References

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM* 54(12):92–103.

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; Ridder, B.; Carrera, A.; Palomeras, N.; Hurtos, N.; and Carreras, M. 2015. ROSPlan: Planning in the robot operating system. In *25th Int. Conf. on Automated Planning and Scheduling*.

Coles, A. J.; Coles, A.; Fox, M.; and Derek, L. 2010. Forward-Chaining Partial-Order Planning. In *International Conference on Automated Planning and Scheduling*.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20.

Gebser, M.; Kaminski, R.; Knecht, M.; and Schaub, T. 2011. plasp: A Prototype for PDDL-Based Planning in ASP. In *International Conference on Logic Programming and Nonmonotonic Reasoning*.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. Clingo = ASP + Control: Preliminary Report. In *Communications of the Thirtieth International Conference on Logic Programming (ICLP)*. (online supplement).

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Wanko, P. 2016. Theory Solving Made Easy with Clingo 5. In *Tech. Comm. 32nd International Conference on Logic Programming (ICLP)*, volume 52 of *OpenAccess Series in Informatics (OASIcs)*. Schloss Dagstuhl.

Gerevini, A., and Long, D. 2005. Plan Constraints and Preferences in PDDL3. Technical report, Dept. of Electronics for Automation, University of Brescia, Italy.

Kaufmann, B.; Leone, N.; Perri, S.; and Schaub, T. 2016. Grounding and Solving in Answer Set Programming. *AI Magazine* 37(3).

Lifschitz, V. 2008. What Is Answer Set Programming? In *Association for the Advancement of Artificial Intelligence*.

Niemueller, T.; Reuter, S.; Ferrein, A.; Jeschke, S.; and Lakemeyer, G. 2015. Evaluation of the RoboCup Logistics League and Derived Criteria for Future Competitions. In *RoboCup Symposium 2015 – Development Track*.

Niemueller, T.; Karpas, E.; Vaquero, T.; and Timmons, E. 2016. Planning Competition for Logistics Robots in Simulation. In *WS on Planning and Robotics (PlanRob) at Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

Niemueller, T.; Lakemeyer, G.; and Ferrein, A. 2013. Incremental Task-level Reasoning in a Competitive Factory Automation Scenario. In *AAAI Spring Symposium - Designing Intelligent Robots: Reintegrating AI*.

Niemueller, T.; Lakemeyer, G.; and Ferrein, A. 2015. The RoboCup Logistics League as a Benchmark for Planning in Robotics. In *WS on Planning and Robotics (PlanRob) at Int. Conf. on Automated Planning and Scheduling (ICAPS)*.